CHAPTER

# NINE

# PETRI NETS AND DATA FLOW

This chapter concerns models inspired by graphs. These models use the nodes of a graph to represent active processing agents that exchange information along paths specified by the edges. We discuss two such models, Petri Nets and Data Flow.

Petri Nets is a formal modeling technique that encodes the states of a dynamic system as the markings of tokens on a graph. Petri Nets expresses in the graph structure the possible state transitions of the system being modeled. Petri Nets has been used to model not only computing systems, but also systems from domains as diverse as neurology, chemistry, and economics. Many theorems have been proved about the formal properties of Petri Nets.

Petri Nets is useful for modeling the states of systems. The tokens passed around a net hold no information. Instead, the number and arrangement of tokens encode the modeled system's state. One can treat a Petri net as a computing mechanism by counting tokens. However, Petri Nets is a weak device for expressing computation. They are not even Turing-equivalent. Data Flow extends the Petri Nets metaphor to associate information with the tokens and to permit computation at the graph nodes. In doing so, Data Flow broadens the limited computational ability of Petri Nets into a mechanism that can compute any computable function. The Data Flow model has excited interest in building *Data Flow machines*, computers whose internal architecture reflects Data Flow's functional nature [Agerwala 82; Treleaven 82]. The designers of these machines hope to avoid the bottleneck of the single-instruction cycle of conventional von Neumann architectures, producing systems that exploit Data Flow's potential concurrency.

Petri Nets is an outgrowth of the 1962 doctoral dissertation of Carl Adam Petri [Petri 62]. Much of the early work on Petri Nets was done by A. W. Holt and his associates at Applied Data Research [Holt 68]. Since then, there has been much interest in both the theory and application of Petri Nets. Peterson's article in *Computing Surveys* [Peterson 77] and his book, *Petri Net Theory and the Modeling of Systems* [Peterson 81], are comprehensive overviews of the subject.

Data Flow models can be traced to Adams' dissertation on data-driven computation [Adams 68], Karp and Miller's work on program schemata [Karp 66], Rodriguez-Bezos's invention of "program graphs" [Rodriguez-Bezos 69], unpublished work by Ivan Sutherland on graphical programming, and the single assignment concept of Tesler and Enea [Tesler 68]. The graph-and-token structure we describe is based on Fosseen's M.I.T. Master's thesis [Fosseen 72]. Early work on Data Flow machines includes the work of Arvind and Gostelow [Arvind 77], Davis [Davis 78], Dennis [Dennis 74], Syre et al. [Berger 82; Comte 79], and Watson and Gurd [Watson 82].

## 9-1 PETRI NETS

Petri Nets models the states of a system by marking the nodes of a graph with tokens. Petri Nets not only represents states by marking nodes, but also encodes the permissible state transitions in the graph structure.

The graph in Figure 9-1 is a Petri net. This graph has two kinds of nodes, *places*, drawn as circles, and *transitions*, drawn as line segments. Directed edges connect the places and the transitions. Every edge connects one place and one transition. There can be several edges between any pair of places and transitions. If there are $k$ edges going from a place to a transition, we say that the *in-degree to* that transition from that place is $k$. Similarly, the *out-degree* from a transition to a place is the number of edges *from* that transition to that place.

**Figure 9-1** A Petri Net.

**Figure 9-2** The barbershop Petri net.

Formally, a Petri net is a bipartite, directed multigraph. That is, a Petri net is like a conventional graph composed of nodes and edges, but (1) the graph is directed—each edge goes *from* a particular node *to* another node, (2) the graph is a multigraph—there can be several edges from any given node to another, and (3) the graph is bipartite—the nodes of the graph can be partitioned into two sets (the transitions and the places), such that each edge connects an element of one set to an element of the other.*

Petri nets are marked with tokens. A *marked* Petri net is the association of a number with each place, the number of tokens on that place. The number of

* There is a natural mapping from Petri nets to multisets. (A *multiset* is a set in which individual elements can appear repeatedly.) In this mapping, the nodes of the graph represent elements of a domain, and the edges of the graph encode functions over multisets of nodes. Peterson presents the details of one such formalization [Peterson 81].

tokens on any place at one time is not bounded but is always finite. Figure 9-2 shows a marked Petri net. This Petri net models the operation of a barbershop. Each token in the *waiting* place represents a customer waiting for a haircut. Each token in the *cutting* place represents a barber giving a haircut. Each token in the *resting* place represents an idle barber. Each token in the *exit* place represents a customer leaving the shop. The figure shows three customers waiting for service, two current haircuts, and one resting barber. In general, the arrangement of tokens on the places of a Petri net represents the state of the system modeled by that net.

**Figure 9-3** States of the barbershop.

Execution of a Petri net is controlled by the number and distribution of tokens on the places of that net. The arrangement of tokens is changed by the *firing* of the transitions. When a transition fires, it removes tokens from the places that have edges running *to* the transition (the input places) and puts tokens on those places that have edges running *from* the transition to them (the output places). More specifically, a transition may fire if it is enabled. A transition is enabled if every place connected to that transition with in-degree $k$ has at least $k$ tokens. For example, let there be three input edges from place p to transition t and five input edges from place q to t. Then t is enabled if p has at least three tokens and q has at least five.

An enabled transition can fire at any time. In firing, a transition consumes one token from (the place associated with) each input edge and produces one token on (the place associated with) each output edge. Thus, firing a transition on a Petri net produces a new marking of the net. Figure 9-3 shows successive firings of transitions of the barbershop net. The successive states of the graph model potential successive states of the (real) barbershop. An enabled transition ceases to be enabled if its enabling tokens are consumed by the firing of some other transition. In Figure 9-4, the firing of transition S disables transition T.

Petri Net models of systems can be used to prove properties such as mutual exclusion (several processes are not engaged in conflicting activities at the same time), liveness (a system of processes does not deadlock; each individual process continues to progress), and reachability (some particular state can or cannot be reached). For example, using the barbershop net we could show that no two barbers ever cut the same customer's hair, that the barbershop as a whole can always make progress in hair cutting, and that the number of customers getting haircuts never exceeds the original number of barbers.

**Figure 9-4** Disabled transition.

## Modeling with Petri Nets

To design a Petri net to model a system, the parts of that system are classified into events and conditions [Peterson 81]. *Events* are actions, such as a barber beginning a haircut, two chemicals combining to form a third, or a job completing on a computer system. *Conditions* are descriptions of the state of a system. Typical conditions include "idle barber," "sodium ion present," and "computer system running." The *preconditions* of an event are those conditions that must be present before an event can occur. Before a barber can begin a haircut, he must be idle and a customer must be waiting. Before a chemical can form in a reaction, the reactants and catalyst must be present. Before a particular kind of computer job completes, the computer system must be running, the job must have started, and it must have been assigned two tape drives. Events remove some preconditions and assert other *postconditions.* The barber beginning a haircut reduces by one the number of idle barbers and waiting customers and produces the postcondition "a barber is cutting." The occurrence of a reaction removes several units of each reactant chemical and produces the postcondition of an additional unit of reaction chemical. The completion of a computer job ensures the postcondition of an idle processor and two more free tape drives.

The usual technique for system modeling with Petri Nets is to make each possible event a transition and each relevant condition a place. If $P$ is a precondition for an event $T$, then the place associated with $P$ is an input place of the transition associated with $T$. If $Q$ is a postcondition of an event $T$, then $Q$'s place is an output place of $T$'s transition. Thus, in the barbershop net the event "a haircut commences" has the preconditions of a waiting customer and an idle barber and the postcondition of a haircut occurring. It consumes one token from each of "waiting customer" and "idle barber," and puts one token on "haircut occurring." Related events can be counted by having several tokens on a place. Without multiple tokens, we would need separate conditions (places) for "one barber idle," "two barbers idle," and "three barbers idle," and would have no way of specifying an unbounded number of waiting customers.

**Mutual exclusion** This technique leads to simple models for many of the standard synchronization problems. For example, consider the problem of modeling the mutual exclusion of $n$ processes. For this problem there are $2n$ events: one event for the entry of each process into its critical region, and another for its exit from its critical region. We recognize $2n + 1$ possible conditions: one condition for each process being "in its critical region," another for it being "in its concurrent region" and one condition that "no process is in its critical region." The preconditions for a process to enter its critical region are that it is "in its concurrent region," and that "no process is in its critical region." The event of a process entering its critical region deletes these two preconditions and asserts the postcondition that the process is "in its critical region." Similarly, a process can exit its critical region when it is "in its critical region." This exit deletes the

**Figure 9-5** Three mutually exclusive processes.

is "in the critical region" precondition, and asserts the postconditions that this particular process is "in its concurrent region" and that "no process is in its critical region." Figure 9-5 shows a Petri net for three mutually exclusive processes. The structure of this net is a straightforward consequence of this analysis. That the net is easier to understand than the English description is a good argument for Petri Nets as a modeling tool.*

This Petri net models three mutually exclusive processes. It is not a program for arranging mutual exclusion. We can analyze this net to show, for example, that the processes are mutually exclusive and do not deadlock. However, the model does not indicate how the mutual exclusion it represents is to be accomplished. Petri nets automatically recognize simultaneous conditions and simultaneously effect changes. This ability avoids the major issue in programming synchronization problems, where the atomic actions are at a much less comprehensive level.

**Dining philosophers** Modeling the dining philosophers problem with Petri Nets is similar to modeling mutual exclusion. Each philosopher cycles through

* We could have modeled this situation without the condition "no process is in its critical region," at the cost of a more complex net. Exercise 9-2 asks for such a net.

six states: thinking; having the left fork in preparation for eating; having both forks; eating; having only the right fork in preparation for thinking; having neither fork; and then back to thinking. For each fork, we have a condition that states that it is free. A philosopher can pick up a fork if the fork is free (and she is ready to do so); a philosopher can drop a fork if she is ready to do so. Figure 9-6 shows a Petri net that models the states of the dining philosophers.

**Figure 9-6** The dining philosophers.

This solution is susceptible to deadlock. Our other examples of dining philosopher programs have avoided deadlock by keeping count of the number of philosophers in the room, barring entry to the fifth philosopher. Exercise 9-4 asks for the Petri net that models the deadlock-free solution.

## Issues in Petri Nets Theory

A major limitation of Petri Nets is an inability to determine if a place is empty. To overcome this problem when modeling, we must allocate explicit places for negated conditions. For example, "this process is not in its critical region" in the mutual exclusion net is a negated condition. However, this is sometimes inadequate to model situations involving counting. If we can bound the number of tokens that can be in a particular place, then we can count the tokens as they leave that place, recognizing when the count is full. But this does not work when the number of tokens that can be on a place is unbounded.

The readers-writers problem illustrates testing for bounded empty places but not for unbounded empty places. In a system with (at most) three readers, a writer can write if three "can read" tokens are present. The Petri net in Figure 9-7 models the readers-writers problem for three readers and a single writer. The limitation of reader-free writing is enforced by three input lines from the "free

**Figure 9-7** The readers-writers problem.

readers" place to the "start writing" transition (and the corresponding three output lines from the "stop writing" transition to the "free readers" place). At any time, the number of tokens in the "free readers" place is the difference between the total number of readers (three) and the number of readers who are reading. A writer can write when the number of readers who are reading is zero—which is equivalent to one token in the "free readers" place for each reader. If there are $n$ readers, we can test for $n$ tokens in the "free readers" place by making the in-degree of the "start writing" transition be $n$. However, this solution is not practical if the number of potential readers is unbounded.

The inability to test for a specific marking (such as zero) in an unbounded place is the essential weakness of Petri nets. The issue of deciding if a given Petri net with a particular marking of tokens can ever achieve some other marking is the *reachability problem.* There are algorithms that solve the reachability problem for Petri Nets [Kosaraju 82; Mayr 81; Sacerdote 77]. These algorithms are equivalent to solving the Petri Nets halting problem. Thus, Petri Nets is not even Turing-equivalent.

## 9-2 DATA FLOW

Conceptually, Data Flow takes the Petri Nets theme of modeling state by successive markings of tokens on a graph structure and transforms it into the idea of expressing computation through successive transformations of the values on the tokens of a graph structure. (This is not to imply that Data Flow was directly derived from Petri Nets.) There are many versions of Data Flow systems, all somewhat similar and all unified by the notion of graph-structured computation. We base our description on the work of Jack Dennis and his colleagues at M.I.T., without prejudice to the many other people who have studied Data Flow.

We get the Data Flow model by performing four transformations on the Petri Nets model. The first of these changes the plain, purely marking tokens of Petri Nets to be holders of data values. Every Data Flow token has some value (such as 5 or true or "Hello") in some specific data type (such as integer or boolean or string) "inscribed" on it.

The second transformation takes the synchronizing transitions of Petri Nets and converts them to the computational primitives. We rename the transitions *actors.* Thus, the Petri Nets transition that merely recognized the presence of tokens on each of its input arcs becomes, say, the Data Flow actor that consumes two tokens, computes the sum of the values on those tokens, and outputs a new token whose value is that sum. By and large, Data Flow retains the Petri Nets idea of consuming the input tokens in firing. However, the Petri Nets allowance of explicit multiple outputs from a single transition is replaced by the Data Flow restriction of each actor to a single output. (Some models of Data Flow do not include this restriction.)

The third change replaces the Petri Nets token holders, the places, with Data Flow links. *Links* are the only things that increase the systemwide number of tokens—a link can branch, placing a copy of its input token on each of its output arms. Petri Nets places can store an unbounded number of tokens. In Data Flow, at most a single token can be on any branch of a link at any time. Certain links are designated input or output links for communication with the external environment. Actors and links are the two kinds of *nodes*.

The final difference between Petri Nets and Data Flow concerns the rules for firing. In Petri Nets, a transition can fire as soon as all its input arcs have tokens. Most Data Flow nodes cannot fire unless all their input arcs have tokens, but there are some significant exceptions. Additionally, since only a single token can rest on any arm of a link at any time, a Data Flow node (actor or link) cannot fire until its output arcs are free of tokens.

Like Petri Nets transitions, Data Flow nodes are enabled when their preconditions are satisfied. They do not have to fire immediately. Instead, an enabled Data Flow node is only guaranteed to fire eventually.

## Data Flow Graphs

*Data Flow graphs* are constructed by connecting actor diagrams with link diagrams. In contrast with Petri Nets, the "circles" in Data Flow do the work, while the "lines" (edges) serve as storage. Data flow graphs distinguish data-carrying paths and control paths. *Data paths* carry the data values of the computation: integers, reals, characters, etc. *Control paths* carry control values (booleans) that "open and close valves," regulating the flow of data around the graph. We draw data items with black (filled-in) tokens and arrowheads and control items with white (open) tokens and arrowheads. In Figure 9-8, we see the two kinds of links: data links and control links. We sometimes take liberties in drawing links to pass many copies of a token to distant places on a graph. A *Data Flow program* is a Data Flow graph with an initial arrangement of tokens on its arcs. A link can fire whenever there is a token on its input arc and its output arcs are all empty. Firing copies the input token to each output arc and consumes the input token. Figure 9-9 shows the firing rules for links.

Data Flow has six kinds of elementary actors: operators, deciders, booleans, constants, gates, and simple-merges. Figure 9-10 illustrates the kinds of elementary actors.

**Figure 9-8** Links.

**Figure 9-9** Firing rules for links.

*Operators* compute primitive functions such as addition and multiplication. An operator can fire when all its input arcs have tokens and its output arc is empty. If the input tokens of an operator have values $v_1, v_2, \ldots, v_n$ and the operator computes the function $f$, then firing the operator consumes these tokens and places a token with value $f(v_1, v_2, \ldots, v_n)$ on the output arc. Figure 9-11 shows the firing of an addition operator.

*Deciders* are the corresponding actors for primitive predicates, such as $\leq$ (less than or equal to). Figure 9-12 shows the firing of decider $>$.

*Boolean actors* (and, or, not) are the boolean functions of their control-value inputs. Their firing organization is the same as that of deciders and operators.

**Figure 9-10** Elementary actors.

**Figure 9-11** Firing an operator.

A *constant actor* produces a stream of its constant as its output and is able to fire whenever its output link is empty. Constant actors have no input arcs.

*Gates* allow control tokens to regulate the flow of data tokens. Data Flow has two types of gates, T (true) gates and F (false) gates. A gate has two inputs, both of which must be present for the gate to fire. The first input is a data token, the second, a control token. If the type of the gate (T or F) matches the data value of the control token, then (in firing) the data token is passed to the output of the gate. If not, the firing gate consumes the data token and no output token is produced. Firing a gate always consumes the control token. Gates are the only Data Flow nodes that do not always produce an output for every set of inputs. Figure 9-13 shows the possible firings of T gates and F gates.

*Simple-merge actors*, like gates, allow control tokens to regulate the flow of data tokens. A simple-merge actor has three input lines: a control line and two data lines—a true line and a false line. In firing, a token is taken from the data line that matches the value on the control line and is placed on the output line. The control token is also consumed. The value on the data line that is not selected is not affected. Hence, each firing of a simple-merge actor consumes exactly two tokens. In contrast with the firing rules for the other elementary actors, there need not be a token on the unselected line for a simple-merge actor to fire. If the control token and the "correct" data token have reached a simple-merge actor, the actor can fire. Figure 9-14 shows the firings of a simple-merge actor.

**Conditionals** One common subgraph of Data Flow programs is to combine a T gate, an F gate, and a simple-merge into a graph that computes a conditional. The Data Flow graph in Figure 9-15 is equivalent to the expression

**Figure 9-12** Firing a decider.

**Figure 9-13** Firings of gates.

**Figure 9-14** Firings of the simple-merge actor.

**Figure 9-15** The Data Flow graph: **if** b **then** ThenVal **else** ElseVal.

**if** b **then** ThenVal **else** ElseVal

In this example it is critical that the unselected gate consume the unneeded token, preventing it from interfering with the next cycle of the computation. If b is true, then ElseVal is consumed by the F gate; if b is false, then ThenVal is consumed by the T gate.

**A simple loop** Figure 9-16 shows a Data Flow graph for computing $z = x^n$, adapted from Dennis [Dennis 74]. This Data Flow graph is equivalent to the pseudoprogram

```
function exp (x, n);
begin
    y := 1;
    i := n;
    while i > 0 do
        begin
            y := y * x;
            i := i − 1;
        end;
    z := y;
    return (z)
end
```

**Figure 9-16** $z = x^n$.

This Data Flow computation is driven by the >0 test. This test sends its boolean signal to each of the three simple-merge actors and the four gates. The simple-merge actors serve only to keep the next input data out of the computation after the first time around the loop. Each of the T gates receives the boolean signal, allowing the data tokens to pass through the computational loop again. One T gate serves to iterate the values of x, another y, and the third i. When i is finally reduced to zero, the boolean signal is false and the F gate passes the answer (the current value of y, called z) to the external environment.

**Demultiplexer** Data Flow diagrams resemble circuit diagrams. This resemblance brings to mind both the circuit designer's repeated use of similar elements and typical circuit design problems. Our next example is a Data Flow demultiplexer (demux) built out of many similar elements. A multiplexer accepts several

**Figure 9-17** A demux unit.

signal lines, typically a power of 2, numbered from 0 through $2^n - 1$, and a control line with an integer in that range, $i$, and passes the value on the $i^{th}$ control line to the output. (Thus, a conditional expression can be viewed as a multiplexer over true and false.) A demultiplexer performs the inverse function—it takes a value $v$ and a control signal $i$ and places $v$ on the $i^{th}$ output line.

We build our demultiplexer out of a tree of demux units. Each unit has two inputs and four outputs. The inputs are the value $\mathsf{v}$ that is to be the eventual output of the demux and the current selection index, $\mathsf{i}$. The four outputs are $\mathsf{v}_{even}$, $\mathsf{i}_{even}$, $\mathsf{v}_{odd}$ and $\mathsf{i}_{odd}$. If $\mathsf{i}$ is an even number (its least significant bit in binary representation is zero), then tokens are sent to the even lines; otherwise, tokens are sent to the odd lines. The output lines $\mathsf{i}_{even}$ and $\mathsf{i}_{odd}$ get the value of $\mathsf{i}$ (integer) divided by 2, ready to have its next bit tested. A demux unit is illustrated in Figure 9-17.

The full demultiplexer tree has one demux unit at the first level, two at the second, four at the next, and so forth. The tokens for the next level of indices (the interior dotted region of Figure 9-17) are omitted on the bottom level. Figure 9-18 shows the structure of the full demultiplexer.*

* This demultiplexer has the disadvantage that its outputs are not in numerical order. If we had tested the high bit of the index at each test, the outputs would be sorted.

**Register** The availability of feedback loops in Data Flow programs leads to the possibility of building registerlike Data Flow graphs. The Data Flow graph in Figure 9-19 behaves like a register. Our archetypical register responds to two kinds of requests: value "setting" requests and value "getting" requests. This register interprets a token with the special symbol s as a request for its value. Any other token stores the value of that token in the register. In either case, the register places a token with its (new) value on the output line.

## Extensions to the Data Flow Model

All Data Flow actors except simple-merge require that all inputs be present before the node is enabled. Even simple-merge requires the presence of the necessary tokens before firing. One extension to Data Flow is the addition of another type of actor, indeterminate-merge [Dennis 77]. An *indeterminate-merge actor* has two inputs. It is enabled when either input line has a token. When an

**Figure 9-18** The demultiplexer structure.

**Figure 9-19** A Data Flow register.

indeterminate-merge actor fires, it passes a token from one of its input lines to its output line, leaving the value on the other line unchanged. A possible firing sequence of an indeterminate-merge actor is shown in Figure 9-20. Our example shows new tokens appearing between the firings. The indeterminate-merge actor is antifair; a token on a particular input line can be arbitrarily and indefinitely ignored.

We can build an $n$-input, indeterminate-merge Data Flow graph by cascading binary indeterminate-merges. Figure 9-21 shows one such graph. Since $n$-input indeterminate-merge graphs are such a straightforward extension of binary indeterminate-merges, we treat the $n$-ary case as primitive.

The addition of indeterminate-merge changes the semantics of Data Flow. Without indeterminate-merge, Data Flow is an applicative, Turing-equivalent formalism like the lambda calculus. Indeterminate-merge introduces indeterminacy. Data Flow programs that use indeterminate-merge are no longer functional. However, indeterminate-merge provides Data Flow with a real-world aspect. Just as hardware devices can handle asynchronous interaction on multiple input lines, a Data Flow system with indeterminate-merge can deal with an asynchronous set of inputs.

**Figure 9-20** Firings of an indeterminate-merge actor.

Data Flow models have been extended to permit a range of possible values for tokens. These extensions include record-structured tokens and tokens that are pointers into a free storage heap.

Our final example of a Data Flow graph brings together several of the elements of the previous examples. We use the feedback idea of the register, the demultiplexer, the indeterminate-merge described above, and structured token values. This example presents a Data Flow graph to solve the airline reservation problem. That problem involves keeping track of the reservations of the seats on an airline flight. Requests come in from travel agents. Each request must be answered, with the answer directed back to the originating agent. The system accepts three different types of requests: reservations, which attempt to reserve seats on the flight; cancellations, which return seats to the flight; and inquiries, which request the number of seats remaining. If there are enough seats left, a reservation of $k$ seats updates the number of seats left (by subtracting $k$) and responds to the agent with $k$. If there are fewer than $k$ seats remaining on the flight, the response is 0. A cancellation of $k$ seats adds $k$ to the number of seats left and responds with $k$; an inquiry leaves the number of seats unchanged, and responds with that number. Specifically, let remaining be the number of seats left on the flight. In the pseudoprogram below, the record accessing functions query-type, size, and agent respectively extract the type of request, the number of seats requested, and the identification number of the inquiring agent.

```
if reservation?(x.query-type) then
    if remaining $\geq$ x.size then
        $\langle$ remaining := remaining $-$ x.size, answer(x.agent) := x.size $\rangle$
    else
        $\langle$ remaining := remaining, answer(x.agent) := 0 $\rangle$
```

```
else if cancellation?(x.query-type) then
        ⟨ remaining := remaining + x.size, answer(x.agent) := x.size ⟩
else if inquiry?(x.query-type) then
        ⟨ remaining := remaining, answer(x.agent) := remaining ⟩
```

In this example, we have paired the consequences of each action. We have done this is because the computation produces two results — one, remaining, to be used in a feedback loop in the program; the other, answer, to be given to the demultiplexer. The demultiplexer takes the agent number and passes this value to that agent's line. The agent's request is on the line labeled x.

Figure 9-22 shows a Data Flow graph of the airline reservation program. In this example, we have used three indeterminate-merges. The one used to receive the requests of the incoming agents must be an indeterminate-merge. The other two could be replaced by several simple actors, at the cost of a more cluttered Data Flow graph. The capacity of the plane, 100 seats, is the value on the initialization token.

**Figure 9-21** Cascading binary indeterminate-merges.

**Figure 9-22** The airline reservation Data Flow graph.

## Perspective

Petri Nets and Data Flow share the conceptual basis of modeling change through successive markings of a graph structure. Petri Nets is a pure modeling technique. Computing with Petri Nets (by tricks such as counting tokens) is difficult. Data Flow transforms the marked, graph-structured machine to a computing formalism. Data Flow captures some quality of concurrency — if several data values reach actors at the same time, then the computations in those actors can be done in parallel.

Data Flow has proved to be a fertile concept. It has been a source for ideas about both computer hardware and programming languages. Several computers whose internal architectures reflect the concepts of Data Flow have been designed and implemented. Two good surveys of Data Flow are a *Computing Surveys* article by Treleaven, Brownbridge, and Hopkins [Treleaven 82] and an *IEEE Computer* issue on Data Flow edited by Agerwala and Arvind [Agerwala 82].

Programming languages have also been developed to express Data Flow computations. One such language is VAL [Ackerman 79; McGraw 82]. Syntactically, VAL looks like an imperative language with the single-assignment rule — programs must guarantee that each variable is assigned only once during program execution. Semantically, VAL is equivalent to a strongly-typed pure Lisp.

## PROBLEMS

**9-1** What would an indeterminate transition in a Petri net be like?

**9-2** Give a Petri net that models the mutual exclusion of three processes using only six places and six transitions.

**9-3** Model a bounded producer-consumer buffer with Petri Nets.

**9-4** Modify the Petri net for the dining philosophers problem so that the system is deadlock-free.

**9-5** Petri Nets can be extended to produce more powerful automata. Peterson ([Peterson 81]) outlines several possible extensions to the standard firing rules: (*a*) *Constraints* modify the firing rule to specify sets of places that must always retain an empty place. (*b*) An *exclusive-or transition* fires when exactly one of its input places has a token. The transition consumes that token. (*c*) *Switch transitions* use the presence or absence of a token on a special "switch" place to determine which output places get tokens. (*d*) *Inhibitor arcs* lead to places that must be empty before a transition can fire. (*e*) *Priorities* can be associated with transitions. If several transitions are enabled, the transition with the highest priority fires. (*f*) *Time Petri Nets* associates two times with each transition. It requires that every enabled transition wait at least as long as its first time but not as long as its second before firing. Each of these extensions allows for determining empty places and each makes Petri Nets Turing-equivalent.

Devise other possible extensions to the firing rules that allow empty place testing and Turing equivalence.

**9-6** Three missionaries and three cannibals come to a river. They want to cross. The only way across the river is in a single rowboat. This boat can hold only two people; it can be rowed by one. Clearly, to cross the river the travelers will have to row back and forth. However, the cannibals are afraid that if some of their group are ever left outnumbered by missionaries (including the people arriving at a river bank) they will be converted, a dire possibility that cannot be allowed to happen.

(a) How can the group cross the river?

(b) Present a Petri net that models this situation, including the constraint on outnumbered cannibals.

**9-7** What is the effect of omitting the T gates in the Data Flow program that computes exponentiation?

**9-8** Write a Data Flow graph that computes successive Fibonacci numbers.

**9-9** What happens to the Data Flow register if the first token sent is the special value S?

**9-10** Redo the Data Flow graph of the airline reservation problem using simple-merges instead of the two unnecessary indeterminate-merges.

## REFERENCES

[**Ackerman 79**] Ackerman, W. B., and J. B. Dennis, "VAL — A Value-Oriented Algorithmic Language: Preliminary Reference Manual," Technical Report TR-218, Computation Structures Group, Laboratory for Computer Science, M.I.T., Cambridge, Massachusetts (June 1979).

[**Adams 68**] Adams, D. A., "A Computation Model with Data Flow Sequencing," Technical Report TR-CS 117, Computer Science Department, Stanford University, Stanford, California (December 1968).

[**Agerwala 82**] Agerwala, T., and Arvind, "Data Flow Systems," *Comput.*, vol. 15, no. 2 (February 1982). Agerwala and Arvind edited this issue of IEEE Computer devoted to Data Flow systems.

[**Arvind 77**] Arvind, and K. P. Gostelow, "A Computer Capable of Exchanging Processors for Time," in B. Gilchrist (ed.), *Information Processing 77: Proceedings of the IFIP Congress 77*, North Holland, Amsterdam (1977), pp. 849–854. Arvind and Gostelow study Data Flow. This paper reports some of their results. The most interesting difference between the Arvind-Gostelow and the Dennis models is that the former allows multiple, simultaneous tokens on a communications line, while the latter does not. "Coloring" of the tokens distinguishes the tokens created by different function invocations.

[**Berger 82**] Berger, P., D. Comte, N. Hifdi, B. Perlois, and J.-C. Syre, "Le Système LAU: Un Multiprocesseur à Assignation Unique," *Tech. Sci. Inf.*, vol. 1, no. 1 (1982). LAU is the French acronym for single-assignment language. This paper and [Comte 79] describe the micro-architecture of a machine for single-assignment computation, a form of Data Flow.

[**Comte 79**] Comte, D., and N. Hifdi, "LAU Multiprocessor: Microfunctional Description and Technological Choices," *Proc. 1st Eur. Conf. Parallel and Distrib. Proc.*, Toulouse, France (February 1979), pp. 8–15.

[**Davis 78**] Davis, A. L., "The Architecture and System Method of DDM1: A Recursively Structured Data Driven Machine," *Proc. 5th Annu. Symp. Comput. Archit.*, IEEE (April 1978), pp. 210–215.

[**Dennis 74**] Dennis, J. B., "First Version of a Data Flow Procedure Language," in B. Robinet (ed.), *Proceedings, Colloque sur la Programmation*, Lecture Notes in Computer Science 19, Springer-Verlag, Berlin (1974), pp. 362–376. This paper is a good description of Data Flow. It includes sections on building structured data objects in Data Flow and coloring Data Flow tokens.

[**Dennis 77**] Dennis, J. B., "A Language for Structured Concurrency," in J. H. Williams, and D. A. Fisher (eds.), *Design and Implementation of Programming Languages*, Lecture Notes in Computer Science 54, Springer-Verlag, Berlin (1977), pp. 231–242. Dennis presents Data Flow at a workshop devoted to developing Ada. In this paper, Dennis describes the indeterminate-merge actor.

[**Fosseen 72**] Fosseen, J. B., "Representation of Algorithms by Maximally Parallel Schemata," Masters thesis, M.I.T., Cambridge, Massachusetts (1972). Fosseen developed the idea of associating values with the tokens on a Data Flow graph.

[**Gordon 81**] Gordon, M., "A Very Simple Model of Sequential Behavior of nMOS," *Proc. VLSI 81 Int. Conf.*, Edinburgh, Scotland (August 1981), pp. 18–21. Gordon gives an example of the similarity of a Data Flow-like applicative system and computer circuitry.

[**Holt 68**] Holt, A. W., H. Saint, R. Shapiro, and S. Warshall, "Final Report of the Information System Theory Project," Report TRRADC-TR-68-305, Rome Air Development Center, Griffiss Air Force Base, New York (1968). The Information System Theory Project was concerned with methods of modeling and evaluating systems. Much of their work (and much of their report) was devoted to Petri Nets.

[**Karp 66**] Karp, R. M., and R. E. Miller, "Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing," *SIAM J. Appl. Math.*, vol. 14, no. 6 (November 1966), pp. 1390–1411. This is a seminal paper on graph-directed computation.

[**Kosaraju 82**] Kosaraju, S. R., "Decidability of Reachability in Vector Addition Systems," *Proc. 14th Annu. ACM Symp. Theory Comp.*, San Francisco (May 1982), pp. 267–281. Kosaraju presents a simpler proof than Mayr [Mayr 81] or Sacerdote [Sacerdote 77] of the decidability of the reachability problem of Petri Nets.

[**Mayr 81**] Mayr, E. W., "An Algorithm for the General Petri Net Reachability Problem," *Proc. 13th Annu. ACM Symp. Theory Comp.*, Milwaukee (May 1981), pp. 238–246. Mayr presents an algorithm for deciding the reachability problem for Petri Nets.

[**McGraw 82**] McGraw, J. R., "The VAL Language: Description and Analysis," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 1 (January 1982), pp. 44–82. McGraw provides an overview of the Data Flow programming language VAL. He argues for special-purpose Data Flow languages for Data Flow machines.

[**Peterson 77**] Peterson, J. L., "Petri Nets," *Comput. Surv.*, vol. 9, no. 3 (September 1977), pp. 223–252. Peterson's paper is a good general survey and tutorial on Petri Nets.

[**Peterson 81**] Peterson, J. L., *Petri Net Theory and the Modeling of Systems*, Prentice-Hall, Englewood Cliffs, New Jersey (1981). This book is a comprehensive description of Petri Nets. It emphasizes the theoretical aspects of Petri Nets.

[**Petri 62**] Petri, C. A., "Kommunikation mit Automaten," Ph.D. dissertation, University of Bonn, Bonn (1962). In his dissertation, Petri lays the groundwork for Petri Net theory. His work is directed at issues of automata theory.

[**Rodriguez-Bezos 69**] Rodriguez-Bezos, J. E., "A Graph Model for Parallel Computation," Report MAC-TR-64, Project MAC, M.I.T., Cambridge, Massachusetts (September 1969). In this doctoral dissertation, Rodriguez-Bezos introduces "program graphs."

[**Sacerdote 77**] Sacerdote, G. S., and R. L. Tenney, "The Decidability of the Reachability Problem for Vector Addition Systems," *Proc. 9th Annu. ACM Symp. Theory Comp.*, Boulder, Colorado (May 1977), pp. 61–76. Sacerdote and Tenney present the first proof of the decidability of the reachability problem for Petri Nets.

[**Tesler 68**] Tesler, L. G., and H. J. Enea, "A Language Design for Concurrent Processes," *Proceedings of the 1968 Spring Joint Computer Conference*, AFIPS Conference Proceedings vol. 32, AFIPS Press, Arlington, Virginia (1968), pp. 403–408. In this paper, Tesler and Enea introduce a single-assignment language called "Compel."

[**Treleaven 82**] Treleaven, P. C., D. R. Brownbridge, and R. P. Hopkins, "Data-Driven and Demand-Driven Computer Architecture," *Comput. Surv.*, vol. 14, no. 1 (March 1982), pp. 93–145. Treleaven et al. survey both Data Flow and Reduction Machine architectures.

[**Watson 82**] Watson, I., and J. Gurd, "A Practical Data Flow Computer," *Comput.*, vol. 15, no. 2 (February 1982), pp. 51–57. Watson and Gurd describe a prototype Data Flow machine.